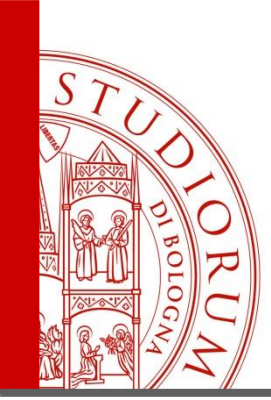


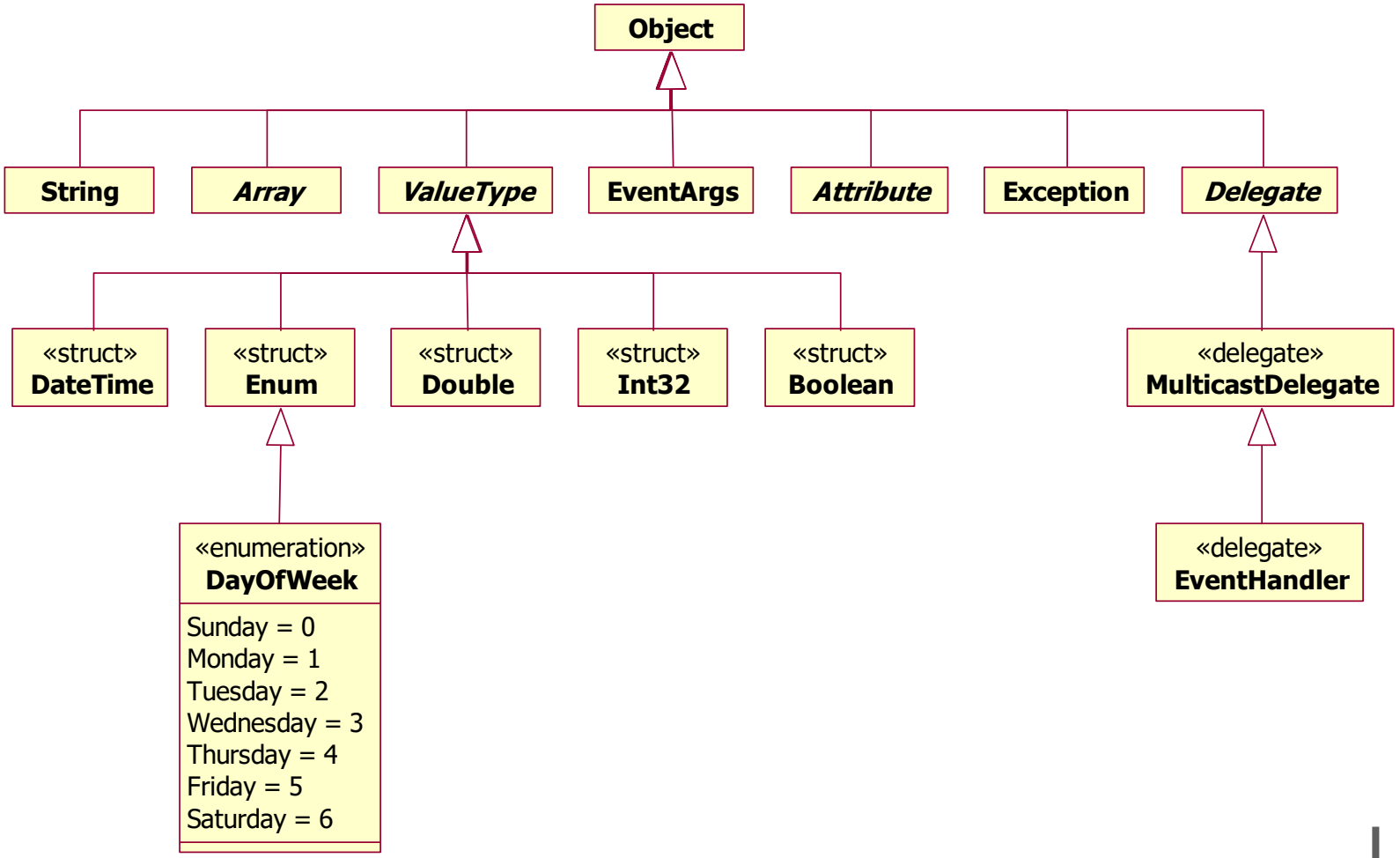
Università degli Studi di Bologna
Corso di Laurea in Ingegneria Informatica

# Classi e Interfacce Base

*Ingegneria del Software T*

**Prof. MARCO PATELLA**
*Dipartimento di Informatica – Scienza e Ingegneria (DISI)*

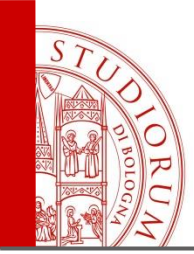
# Framework .NET: Overview



# System.Object

- La radice della gerarchia dei tipi
  - tutte le classi nel framework.NET Framework sono derivate da **`Object`**
- Ogni metodo definito nella classe **`Object`** è disponibile in tutti gli oggetti del sistema

| **Object** |
| --- |
| + Object ( )<br># Finalize ( )<br>+ GetHashCode ( ) : System.Int32<br>+ Equals ( [in] obj : System.Object ) : System.Boolean<br>+ ToString ( ) : System.String<br>+ Equals ( [in] objA : System.Object , [in] objB : System.Object ) : System.Boolean<br>+ ReferenceEquals ( [in] objA : System.Object , [in] objB : System.Object ) : System.Boolean<br>+ GetType ( ) : System.Type<br># MemberwiseClone ( ) : System.Object |

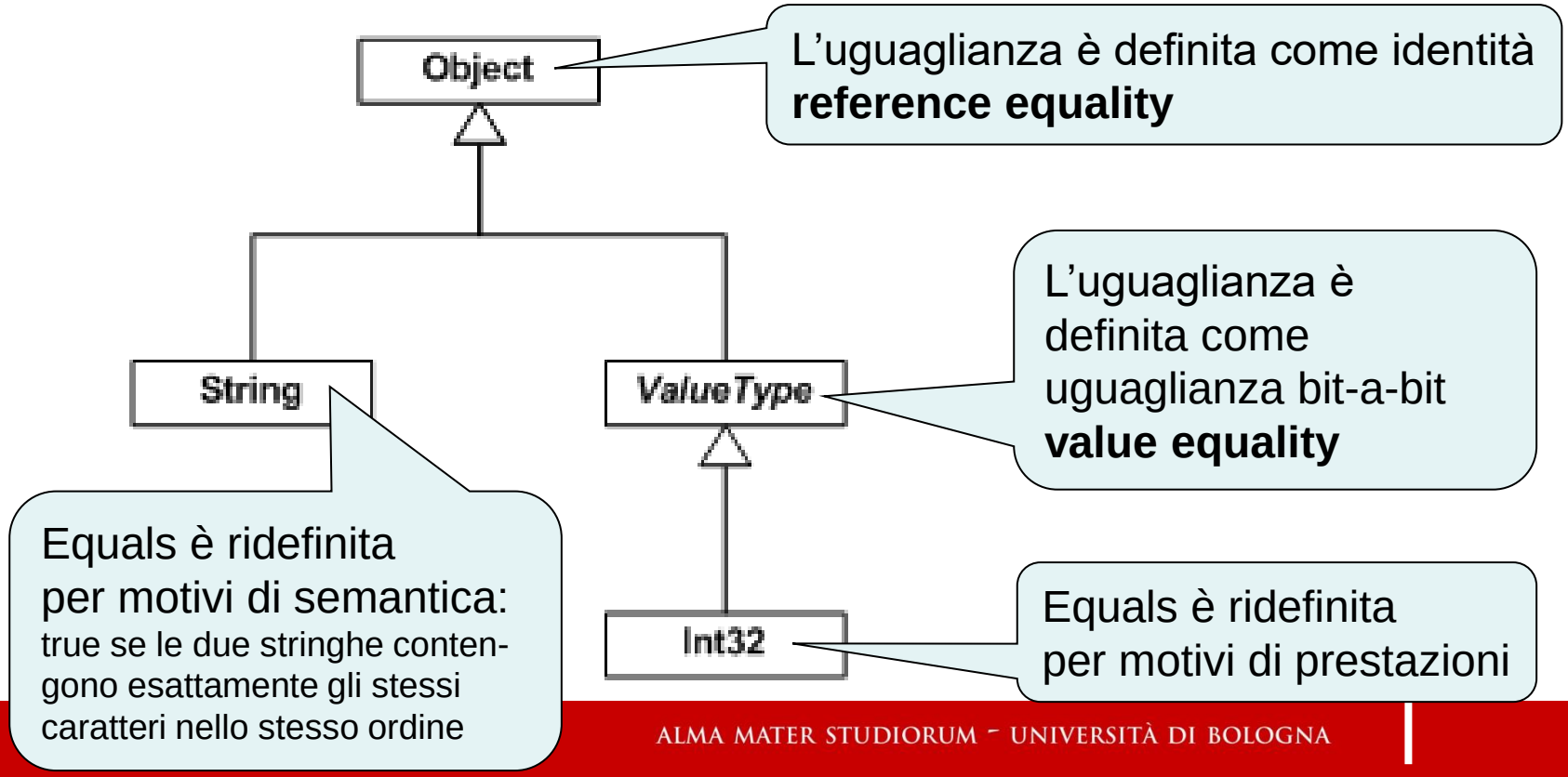


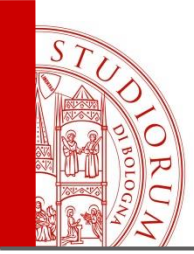
# System.Object

- Le classi derivate possono (e/o devono) sovrascrivere (override) alcuni di tali metodi, tra cui:
  - **`Equals`** – Supporta il confronto tra oggetti
  - **`ToString`** – Crea una stringa “comprensibile” che descrive un’istanza della classe
  - **`GetHashCode`** – Genera un numero corrispondente al valore (stato) dell’oggetto per consentire l’uso di tabelle hash
  - **`Finalize`** – Effettua operazioni di “pulizia” prima che un oggetto venga automaticamente distrutto

# Object.Equals

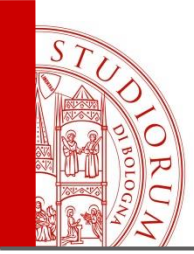
- **`public virtual bool Equals(object obj);`**
- Valore di ritorno: **`true`** se **`this`** è uguale a **`obj`**, altrimenti **`false`**





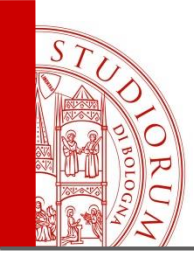
# Object.Equals

- Le seguenti condizioni devono essere **`true`** per tutte le implementazioni del metodo **`Equals`** Nell'elenco, **`x`**, **`y`**, e **`z`** rappresentano riferimenti non nulli a oggetti
  - **`x.Equals(x)`** restituisce **`true`**
  - **`x.Equals(y)`** restituisce lo stesso valore di **`y.Equals(x)`**
  - **`x.Equals(y)`** restituisce **`true`** se sia **`x`** che **`y`** sono **`NaN`**
  - **`(x.Equals(y) && y.Equals(z))`** restituisce **`true`** se e solo se **`x.Equals(z)`** restituisce **`true`**
  - Chiamate successive a **`x.Equals(y)`** restituiscono sempre lo stesso valore fintantoché gli oggetti rteferenziati da **`x`** e **`y`** non vengono modificati
  - **`x.Equals(`**un riferimento **`null)`** restituisce **`false`**
- Le implementazioni di **`Equals`** non devono generare eccezioni



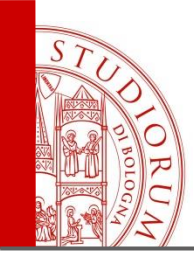
# Object.Equals

- I tipi che sovrascrivono **`Equals`** devono anche sovrascrivere **`GetHashCode`**; altrimenti, **`Hashtable`** potrebbe non funzionare correttamente
- Se il linguaggio permette l'overloading di operatori e, per un certo tipo, si effettua l'overloading dell'operatore di uguaglianza, tale tipo deve anche sovrascrivere il metodo **`Equals`**
  Tale implementazione del metodo **`Equals`**
  **deve restituire gli stessi risultati dell'operatore di uguaglianza**



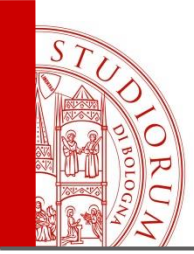
# Object.Equals

```
public class Point
{
  private readonly int _x, _y;
  …
  public override bool Equals(object obj)
  {
    //Check for null and compare run-time types.
    if(obj == null || GetType() != obj.GetType())
       return false;
    Point p = (Point) obj;
    return (_x == p._x) && (_y == p._y);
  }
  public override int GetHashCode()
  {
    return _x ^ _y;
  }
}
```



# Object.Equals

```
public class SpecialPoint : Point
{
  private readonly int _w;
  …
  public SpecialPoint(int x, int y, int w) : base(x, y)
  {
    _w = w;
  }
  public override bool Equals(object obj)
  {
    return base.Equals(obj) &&
      _w == ((SpecialPoint) obj)._w;
  }
  public override int GetHashCode()
  {
    return base.GetHashCode() ^ _w;
  }
}
```



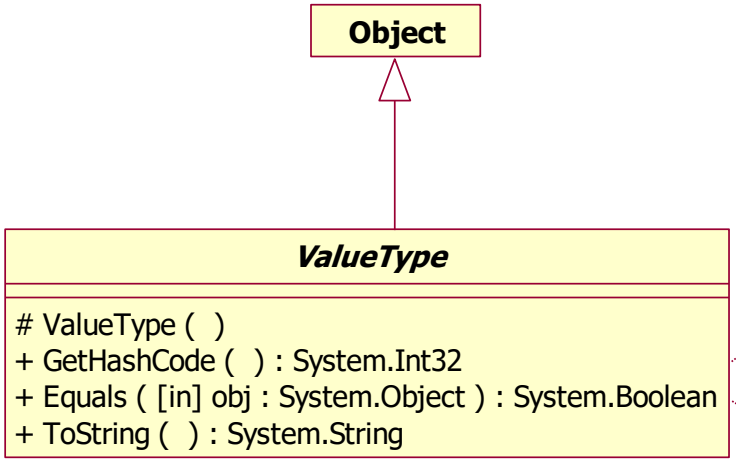
# Object.Equals

```
public class Rectangle
{
  private readonly Point _a, _b;
  …
  public override bool Equals(object obj)
  {
    if(obj == null || GetType() != obj.GetType())
      return false;
    Rectangle r = (Rectangle) obj;
    // Uses Equals to compare variables.
    return _a.Equals(r._a) && _b.Equals(r._b);
  }
  public override int GetHashCode()
  {
    return _a.GetHashCode() ^ _b.GetHashCode();
  }
}
```

# Object.Equals

```
public struct Complex
{
  private readonly double _re, _im;
  …
  public override bool Equals(object obj)
  {
    return obj is Complex && this == (Complex) obj;
  }
  public override int GetHashCode()
  {
    return _re.GetHashCode() ^ _im.GetHashCode();
  }
  public static bool operator ==(Complex x, Complex y)
  {
    return x._re == y._re && x._im == y._im;
  }
  public static bool operator !=(Complex x, Complex y)
  {
    return !(x == y);
  }
}
```

# System.ValueType

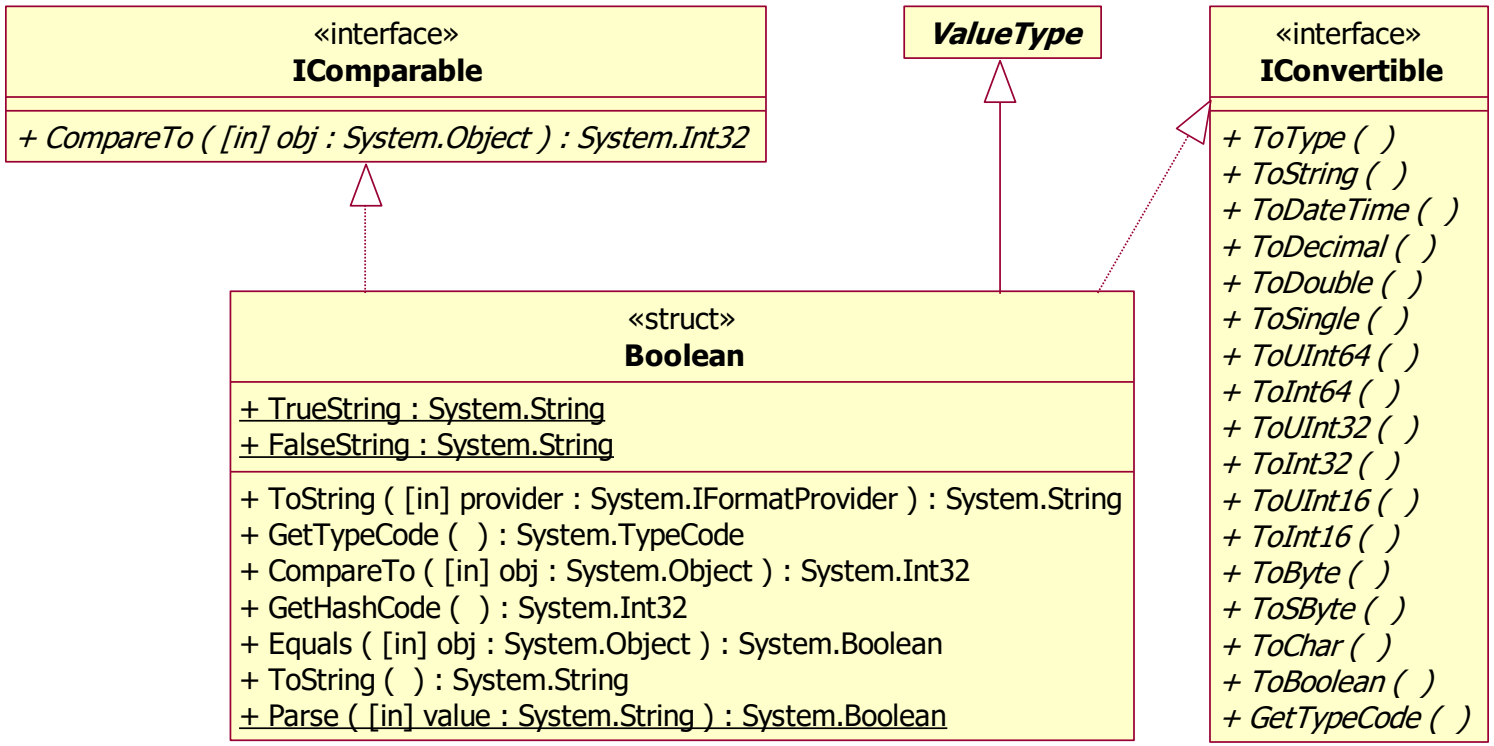



One or more fields of the derived type is used to calculate the return value. If one or more of those fields contains a mutable value, the return value might be unpredictable, and unsuitable for use as a key in a hash table. In that case, consider writing your own implementation of GetHash Code that more closely represents the concept of a hash code for the type.

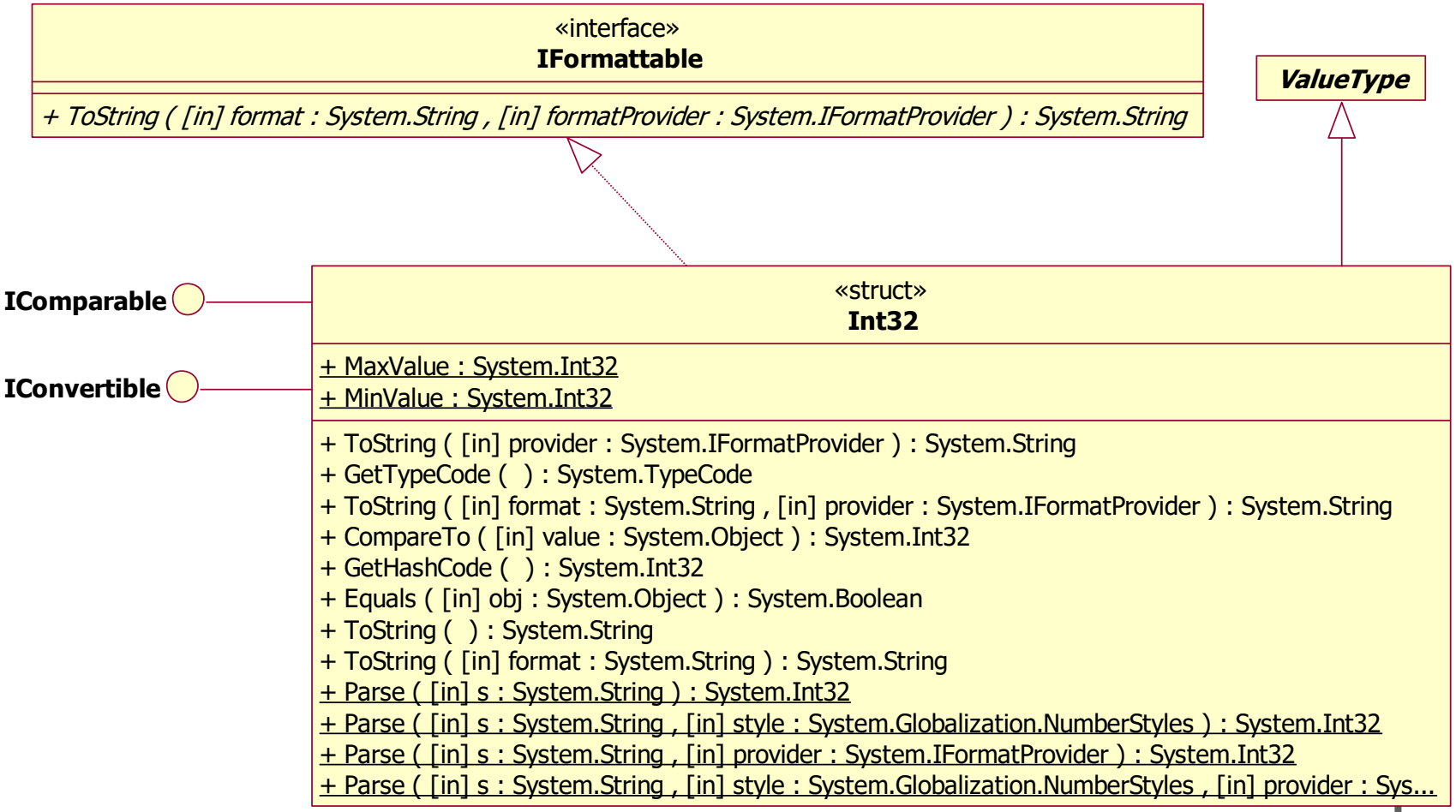
The default implementation of the Equals method uses reflection to compare the corresponding fields of obj and this instance. Override the Equals method for a particular type to improve the performance of the method and more closely represent the concept of equality for the type.

**Esempio 1.0**

# System.Boolean

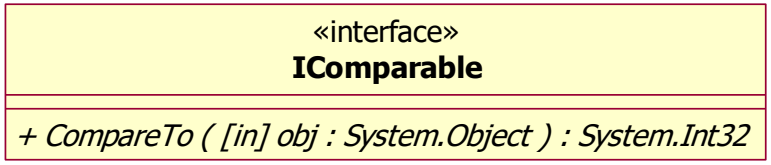


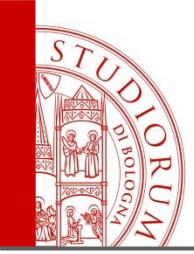
# System.Int32



# System.IComparable

- Confronta l'istanza corrente con un altro oggetto dello stesso tipo
- **Valore di ritorno**: un 32-bit signed integer che indica l'ordine relativo degli oggetti confrontati
- La semantica del valore restituito è la seguente:
  - Minore di zero – l'istanza **`this`** precede **`obj`**
  - Zero – l'istanza **`this`** è uguale a **`obj`**
  - Maggiore di zero - l'istanza **`this`** segue **`obj`**
- Per definizione, ogni oggetto segue un riferimento **`null`**
- Il parametro **`obj`** deve essere dello stesso tipo della classe o value type che implementa questa interfaccia; altrimenti, va lanciata una **`ArgumentException`**

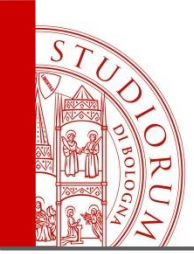




# System.IComparable

- Note per gli Implementatori:
  Per ogni oggetto **`A`**, **`B`** e **`C`**, devono valere le seguenti condizioni:
  - **`A.CompareTo(A)`** deve restituire zero
  - Se **`A.CompareTo(B)`** restituisce zero
    allora anche **`B.CompareTo(A)`** deve restituire zero
  - Se **`A.CompareTo(B)`** restituisce zero e **`B.CompareTo(C)`**
    restituisce zero allora anche **`A.CompareTo(C)`** deve restituire zero
  - Se **`A.CompareTo(B)`** restituisce un valore diverso da zero
    allora **`B.CompareTo(A)`** deve restituire un valore dal segno opposto
  - Se **`A.CompareTo(B)`** restituisce un valore x diverso da zero,
    e **`B.CompareTo(C)`** un valore y dello stesso segno di x,
    allora **`A.CompareTo(C)`** deve restituire un valore dello stesso segno di x e y

**Esempio 1.1a**

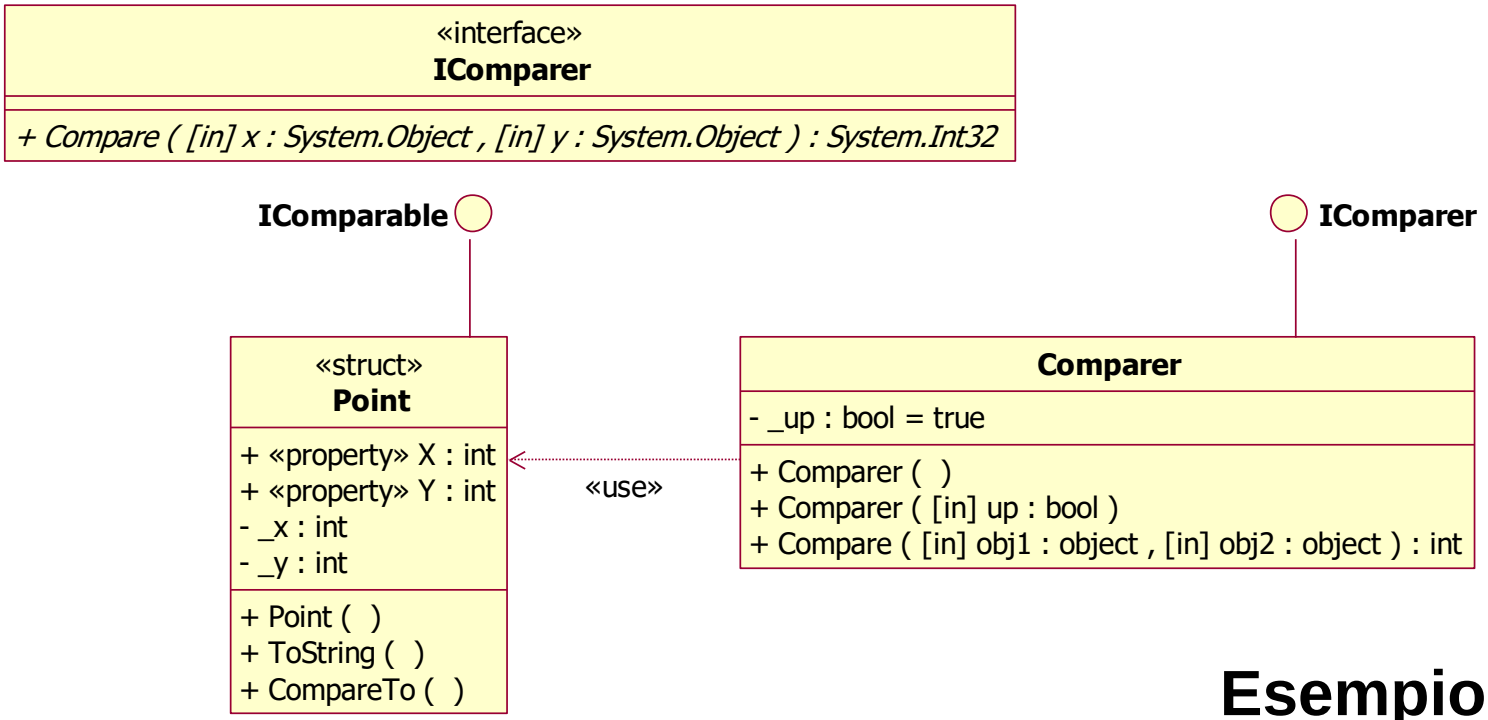


# System.IComparable

- Se volessi:
  - Ordinare i punti in ordine decrescente
  - Ordinare dei film
    - Per genere, oppure
    - Per titolo
  - Ordinare degli studenti
    - Per cognome e nome, oppure
    - Per matricola, oppure
    - Per corso di studio
  - …

# System.Collections.IComparer

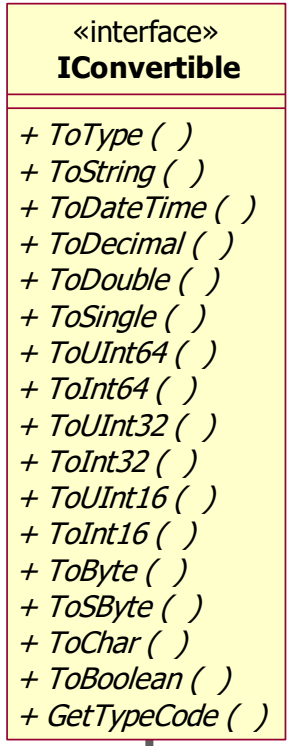
- Questa interfaccia è usata, ad esempio, dai metodi **`Array.Sort`** e **`Array.BinarySearch`**
- Fornisce un modo per personalizzare il criterio di ordinamento di una collezione




**Esempio 1.1b**

# System.IConvertible

- Questa interfaccia fornisce metodi per convertire il valore di un'istanza di un tipo che implementa l'interfaccia in un valore equivalente di un tipo CLR
- I tipi **CLR** sono **`Boolean`**, **`SByte`**, **`Byte`**, **`Int16`**, **`UInt16`**, **`Int32`**, **`UInt32`**, **`Int64`**, **`UInt64`**, **`Single`**, **`Double`**, **`Decimal`**, **`DateTime`**, **`Char`**, e **`String`**
- Se non esiste una conversione sensata verso un tipo CLR, l'implementazione particolare del corrispondente metodo dell'interfaccia deve lanciare una **`InvalidCastException`**
  - Ad esempio, questa interfaccia è implementata dal tipo **`Boolean`**; l'implementazione del metodo **`ToDateTime`** lancia un'eccezione in quanto non esiste alcun valore **`DateTime`** equivalente a un valore del un tipo **`Boolean`**



# System.Convert

| Convert |
| --- |
| + DBNull : System.Object |
| + GetTypeCode ( )<br>+ IsDBNull ( )<br>+ ChangeType ( )<br>+ ToBoolean ( )<br>+ ToChar ( )<br>+ ToSingle ( )<br>+ ToDouble ( )<br>+ ToDecimal ( )<br>+ ToDateTime ( )<br>+ ToByte ( )<br>+ ToSByte ( )<br>+ ToInt16 ( )<br>+ ToUInt16 ( )<br>+ ToInt32 ( )<br>+ ToUInt32 ( )<br>+ ToInt64 ( )<br>+ ToUInt64 ( )<br>+ ToString ( )<br>+ ToBase64String ( )<br>+ ToBase64String ( )<br>+ FromBase64String ( )<br>+ ToBase64CharArray ( )<br>+ FromBase64CharArray ( ) |

- In **`System.Int32`**, l'implementazione dell'interfaccia **`System.IConvertible`** è un esempio di "*explicit interface implementation*":

```
int x = 32;
double d = x.ToDouble(…); // No!
```

  È necessario scrivere:

```
((IConvertible) x).ToDouble(…)
```

- Se necessario, utilizzare la classe **`Convert`**:

```
Convert.ToDouble(x)
```

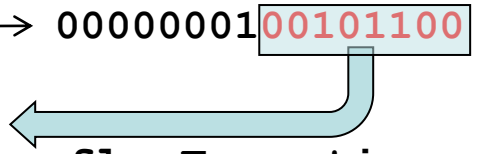
# System.Convert

- Lancia un'eccezione se la conversione non è supportata

```
bool b = Convert.ToBoolean(DateTime.Today);
  //  InvalidCastException
```

- Effettua **conversioni controllate**

```
int k = 300;
byte b = (byte) k;  //  b == 44
byte b = Convert.ToByte(k);  //  OverflowException
```

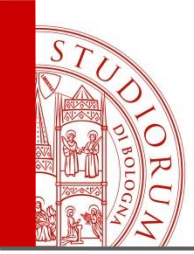



- In alcuni casi, esegue un arrotondamento:

```
double d = 42.72;
int k = (int) d; // k == 42
int k = Convert.ToInt32(d); // k == 43
```

- È utile anche quando si ha una **string** che deve essere convertita in valore numerico:

```
string myString = "123456789";
int myInt = Convert.ToInt32(myString);
```



# Conversione di tipo

- **Una conversione di ampliamento** avviene quando un valore di un tipo viene convertito verso un altro tipo che è di dimensione uguale o superiore
  - Da **`Int32`** a **`Int64`**
  - Da **`Int32`** a **`UInt64`**
  - Da **`Int32`** a **`Single`** (con possibile perdita di precisione)
  - Da **`Int32`** a **`Double`**
- **Una conversione di restrizione** avviene quando un valore di un tipo viene convertito verso un altro tipo che è di dimensione inferiore
  - Da **`Int32`** a **`Byte`**
  - Da **`Int32`** a **`SByte`**
  - Da **`Int32`** a **`Int16`**
  - Da **`Int32`** a **`UInt16`**
  - Da **`Int32`** a **`UInt32`**

# Conversione di tipo

- **Conversioni implicite** – non generano eccezioni
  - **Conversioni numeriche**
    Il tipo di destinazione dovrebbe essere in grado di contenere,
    senza perdita di informazione, tutti i valori ammessi dal tipo di partenza
    Eccezione:

```
int k1 = 1234567891;
float b = k1;
int k2 = (int) b;  //  k2 == 1234567936
```

  - **Up cast**
    **Principio di sostituibilità**: deve sempre essere possibile utilizzare una classe derivata al posto della classe base

```
B b = new B(…);  //  class B : A
A a = b;
```

# Conversione di tipo

- **Conversioni esplicite** – possono generare eccezioni
  - **Conversioni numeriche**
    Il tipo di destinazione non sempre è in grado di contenere il valore del tipo di partenza

```
int k1 = -1234567891;
uint k2 = (uint) k1;  //  k2 == 3060399405

int k1 = -1234567891;
uint k2 = checked((uint) k1);  //  OverflowException

int k1 = -1234567891;
uint k2 = Convert.ToUInt32(k1);  // OverflowException
```

# Conversione di tipo

- **Conversioni esplicite** – possono generare eccezioni
  - **Down cast**

```
A a = new B(…);  //  class B : A
B b = (B) a;  //  Ok

a = new A(…);
b = (B) a;  //  InvalidCastException

if(a is B)  //  if(a.GetType() == typeof(B))
{
  b = (B) a;  //  Non genera eccezioni
  …
}

b = a as B;  //  b = (a is B) ? (B) a : null;
if(b != null)
{
  …
}
```

# Conversione di tipo

- **Boxing** – **up cast** (conversione implicita)

```
int k1 = 100;
object o = k1;  //  Copia!
k1 = 200;
```

- **Unboxing** – **down cast** (conversione esplicita)

```
int k2 = (int) o;  //  k1 = 200, k2 = 100

double d1 = (double) k1;  //  Ok
d1 = k1;  //  Ok
d1 = o;  //  Non compila!
d1 = (double) o;  //  InvalidCastException
d1 = (int) o;  //  Ok
```

# Conversione di tipo definite dall'utente

```
public static implicit operator typeOut(typeIn obj)
public static explicit operator typeOut(typeIn obj)
```

- Metodi statici di una classe o di una struttura
- La *keyword* **`implicit`** indica l'utilizzo automatico (cast implicito)
  Il metodo non deve generare eccezioni
- La *keyword* **`explicit`** indica la necessità di un cast esplicito
  Il metodo può generare eccezioni
- **`typeOut`** è il tipo del risultato del cast
- **`typeIn`** è il tipo del valore da convertire
- **`typeIn`** o **`typeOut`** deve essere il tipo che contiene il metodo

**Esempio 1.2**

# Conversioni a string

- Conversioni a **`string`** (di un **`Int32`**):
  - **`ToString()`**

```
int k1 = -1234567891;
string str = k1.ToString(); // str == "-1234567891"
```

  - **`ToString(string formatString)`**
    l'istanza è formattata secondo il **`NumberFormatInfo`** dell'impostazione cultura (*culture*) corrente

```
k1.ToString("X"); // = "B669FD2D"
k1.ToString("C"); // = "-€ 1.234.567.891,00"
k1.ToString("C0"); // = "-€ 1.234.567.891"
k1.ToString("N0"); // = "-1.234.567.891"
k1.ToString("E"); // = "-1,234568E+009"
```

**Esempio 1.3**

# Conversioni a string

- Conversioni a **`string`** (di un **`Int32`**):
  - **`String.Format(string format, params object[] args)`**

    Il parametro **`format`** è costituito da uno o più elementi di formato nella forma: **`{index[,alignment][:formatString]}`**

```
int k1 = -1234567891;

String.Format("{0}",k1); // = "-1234567891"
String.Format("{0:X}",k1); // = "B669FD2D"
String.Format("{0,5:X}",k1); // = "B669FD2D"
String.Format("{0,10:X}",k1); // = "△△B669FD2D"
String.Format("{0,-10:X}",k1); // = "B669FD2D△△"
String.Format("{0:N0}",k1); // = "-1.234.567.891"
```

# Conversioni da string

- Conversioni da **`string`** (in un **`Int32`**):
  - **`Int32.Parse(string str)`**

```
Int32.Parse("-1234567891"); // -1234567891
Int32.Parse("-1.234.567.891"); // FormatException
Int32.Parse(""); // FormatException
Int32.Parse("-1234567891999"); // OverflowException
Int32.Parse(null); // ArgumentNullException
```

  - **`Int32.Parse(string str, System.Globalization.NumberStyles style)`**

    **`NumberStyles`** determina lo stile permesso per i parametri stringa passati ai metodi **`Parse`** delle classi basi numeriche

# Conversioni da string

«enumeration»
**NumberStyles**

None = 0
AllowLeadingWhite = 1
AllowTrailingWhite = 2
AllowLeadingSign = 4
AllowTrailingSign = 8
AllowParentheses = 16
AllowDecimalPoint = 32
AllowThousands = 64
AllowExponent = 128
AllowCurrencySymbol = 256
AllowHexSpecifier = 512
Integer = 7
HexNumber = 515
Number = 111
Float = 167
Currency = 383
Any = 511

- I simboli da usare per la valuta, il separatore delle migliaia, il separatore decimale e il simbolo del segno sono specificati da **`NumberFormatInfo`**
- Gli attributi di **`NumberStyles`** sono indicati usando l'OR (inclusivo) bit-a-bit dei vari flag di campo

```
Int32.Parse("-1.234.567.891",
   System.Globalization.NumberStyles.Number); // ok
Int32.Parse("B669FD2D",
   System.Globalization.NumberStyles.HexNumber); // ok
```

# Conversioni a/da string

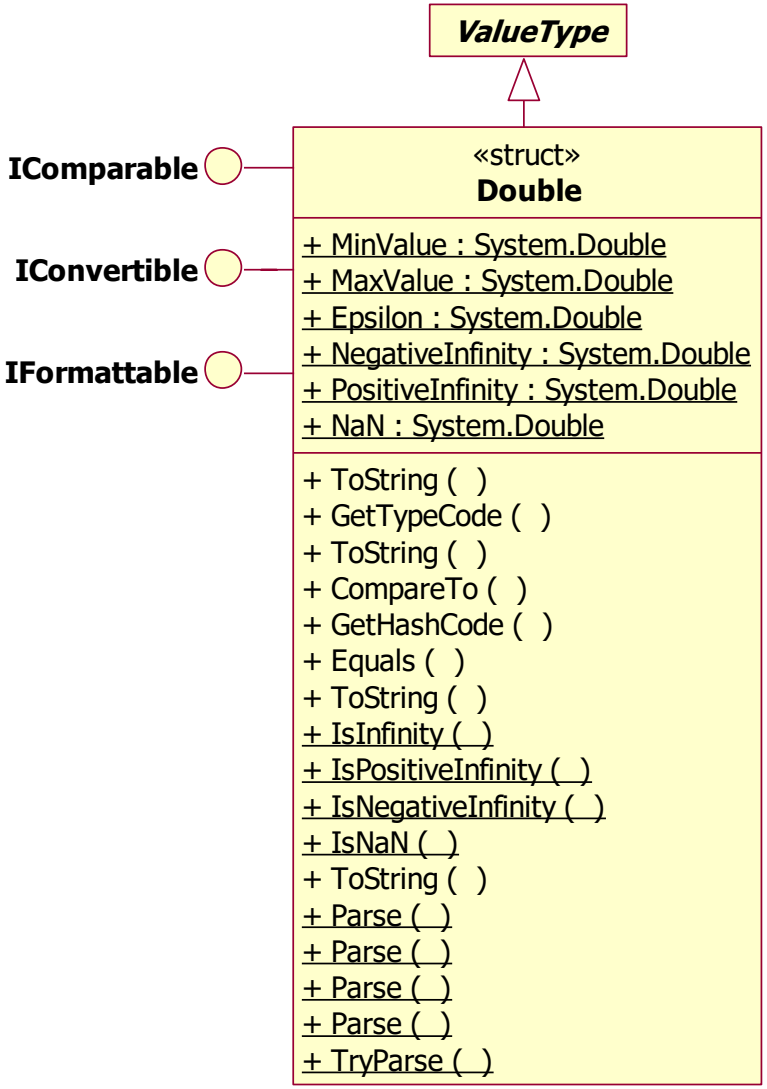
- Conversioni a **`string`** (di un **`Int32`**):
  - **`Convert.ToString(int value, int toBase)`**
    **`toBase`** = 2, 8, 10, 16

```
int k1 = -1234567891;
Convert.ToString(k1); // "-1234567891"
Convert.ToString(k1,10); // "-1234567891"
Convert.ToString(k1,16); // "b669fd2d"
```

- Conversioni da **`string`** (in un **`Int32`**):
  - **`Convert.ToInt32(string str, int fromBase)`**
    **`fromBase`** = 2, 8, 10, 16

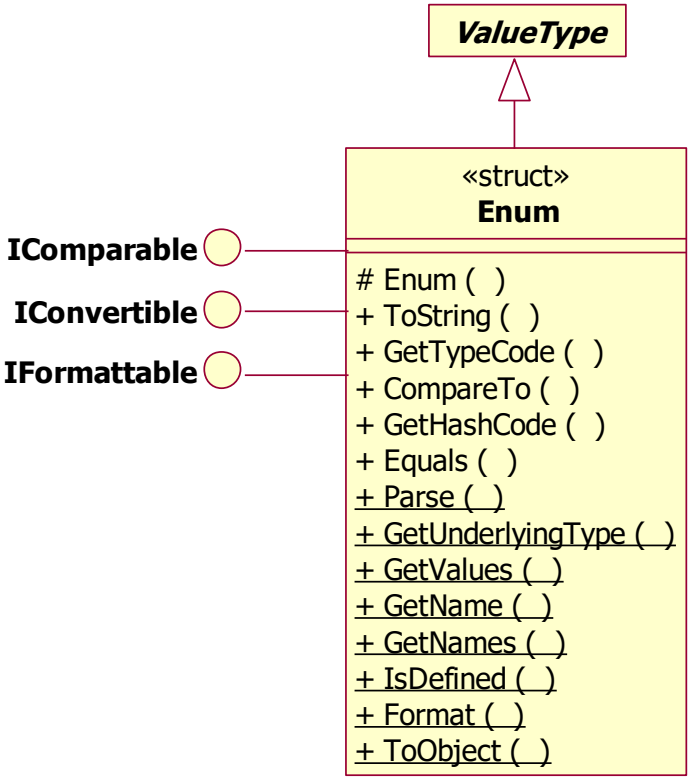
```
Convert.ToInt32("-1234567891"); // -1234567891
Convert.ToInt32("-1234567891",10); // -1234567891
Convert.ToInt32("B669FD2D",16); // -1234567891
Convert.ToInt32("0xB669FD2D",16); // -1234567891
Convert.ToInt32("B669FD2D",10); // FormatException
```

# System.Double




- Segue le specifiche IEEE 754
- Supporta ± 0, ± Infinity, NaN
- **`Epsilon`** rappresenta il più piccolo **`Double`** positivo > 0
- Il metodo **`TryParse`** è analogo al metodo **`Parse`**, ma non lancia eccezioni in caso di fallimento della conversione
  - Se la conversione ha successo, il valore di ritorno è **`true`** e il parametro di uscita è posto pari al risultato della conversione
  - Se la conversione fallisce, il valore di ritorno è **`false`** e il parametro di uscita è posto pari a zero

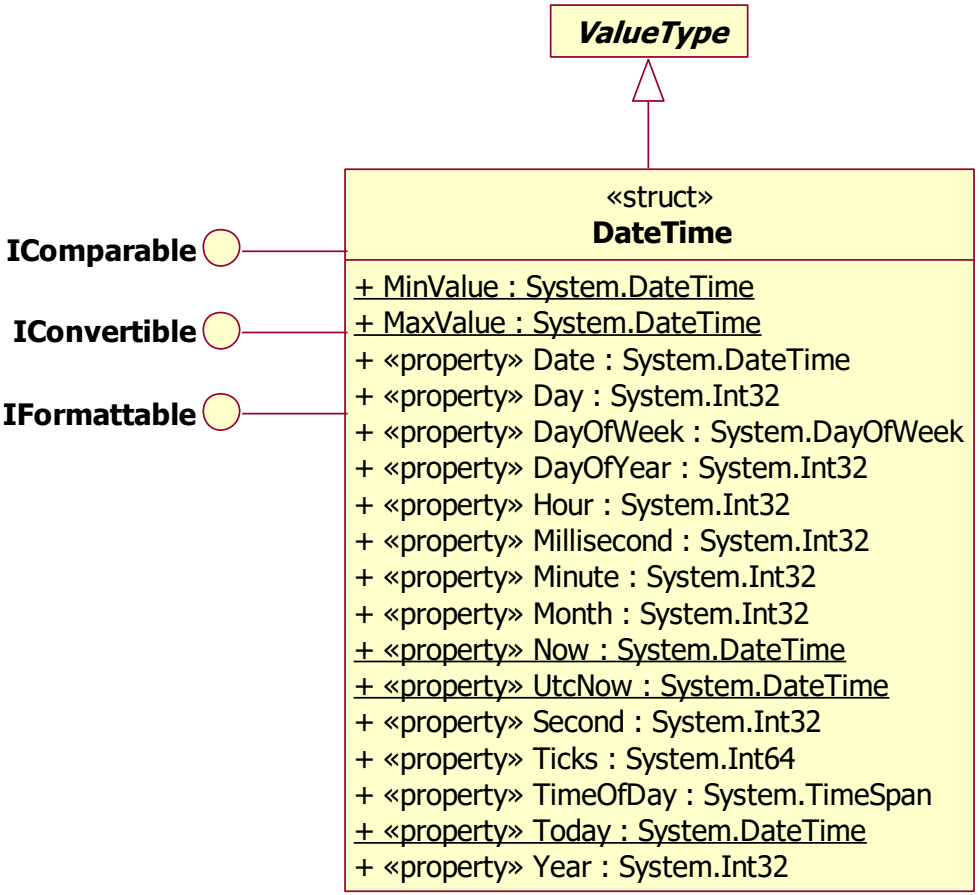
# System.Enum




- **`Enum`** fornisce metodi per
  - Confrontare istanze di questa classe
  - Convertire il valore di un'istanza nella sua rappresentazione a stringa
  - Convertire la rappresentazione a stringa di un numero in un'istanza della classe
  - Creare un'istanza di un'enumerazione e valore specifico
- È possibile trattare un **`Enum`** come un campo bit

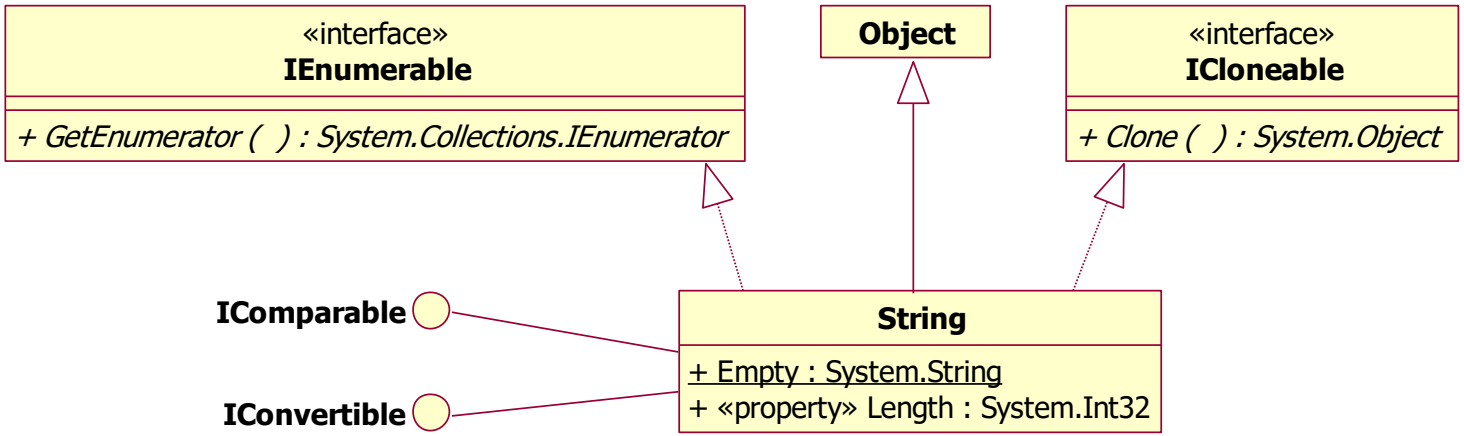
**Esempio 1.4**

# System.DateTime




- Rappresenta un istante nel tempo, tipicamente espresso come data e ora del giorno
- Il tipo di valore **`DateTime`** rappresenta le date e le ore con valori che vanno dalla mezzanotte del 1 gennaio 0001 d.C. (Era Comune) alle 11:59:59 P.M., 31 dicembre 9999 d.C.
- I valori temporali sono misurati in unità di 100 ns chiamate tick
- **`DateTime`** rappresenta un istante nel tempo, mentre **`TimeSpan`** rappresenta un intervallo di tempo

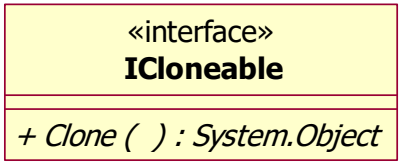
# System.String




- Una stringa immutabile di caratteri Unicode a lunghezza
- Una **`String`** è detta immutabile perché, una volta creata, il suo valore non può essere modificato
- I metodi che sembrano modificare una **`String`**, in realtà restituiscono una nuova **`String`** contenente la modifica
- Se è necessario modificare il contenuto effettivo di un oggetto tipo stringa, utilizzare la classe **`System.Text.StringBuilder`**

**Esempio 1.5**

# System.ICloneable




- Supporta la clonazione, che crea una nuova istanza di una classe con lo stesso stato di un'istanza esistente

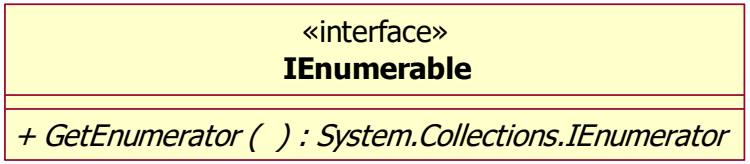
- **`Clone`** crea un nuovo oggetto che è una copia dell'istanza corrente
- **`Clone`** può essere implementato come:
  - una **copia superficiale** (*shallow*), vengono duplicati solo gli oggetti di primo livello, non vengono create nuove istanze di alcun campo

```
public object Clone()
{
   return MemberwiseClone();
}
```

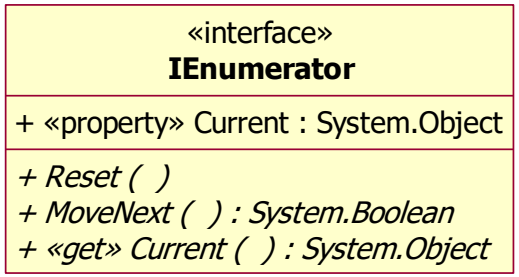
  - una **copia profonda** (deiep), tutti gli oggetti vengono duplicati
- **`Clone`** restituisce una nuova istanza dello stesso tipo (o eventualmente un tipo derivato) dell'oggetto corrente

# System.Collections.IEnumerable

- **`GetEnumerator`** restituisce un enumeratore che può essere utilizzato per scorrere una collezione




- Espone l'enumeratore, che supporta una semplice iterazione su una collezione




- Gli enumeratori permettono solamente di **leggere** i dati della collezione
- Gli enumeratori non possono essere usati per modificare la collezione

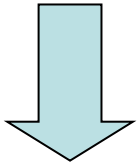
- **`Reset`** riporta l'enumeratore allo stato iniziale
- **`MoveNext`** si sposta all'elemento successivo, restituendo
  - **`true`** se l'operazione ha successo
  - **`false`** se l'enumeratore ha oltrepassato l'ultimo elemento
- **`Current`** restituisce l'oggetto attualmente referenziato

# System.Collections.IEnumerator

- Non deve essere implementata direttamente da una classe contenitore
- Deve essere implementata da una classe separata (eventualmente annidata nella classe contenitore) che fornisce la funzionalità di iterare sulla classe contenitore
- Tale suddivisione di responsabilità permette di utilizzare contemporaneamente più enumeratori sulla stessa classe contenitore
- La classe contenitore deve implementare l'interfaccia **`IEnumerable`**
- Se una classe contenitore viene modificata, tutti gli enumeratori a essa associati vengono invalidati e non possono più essere utilizzati (**`InvalidOperationException`**)

# System.Collections.IEnumerator

```
IEnumerator enumerator = enumerable.GetEnumerator();
while (enumerator.MoveNext())
{
  MyType obj = (MyType) enumerator.Current;
  …
}
```



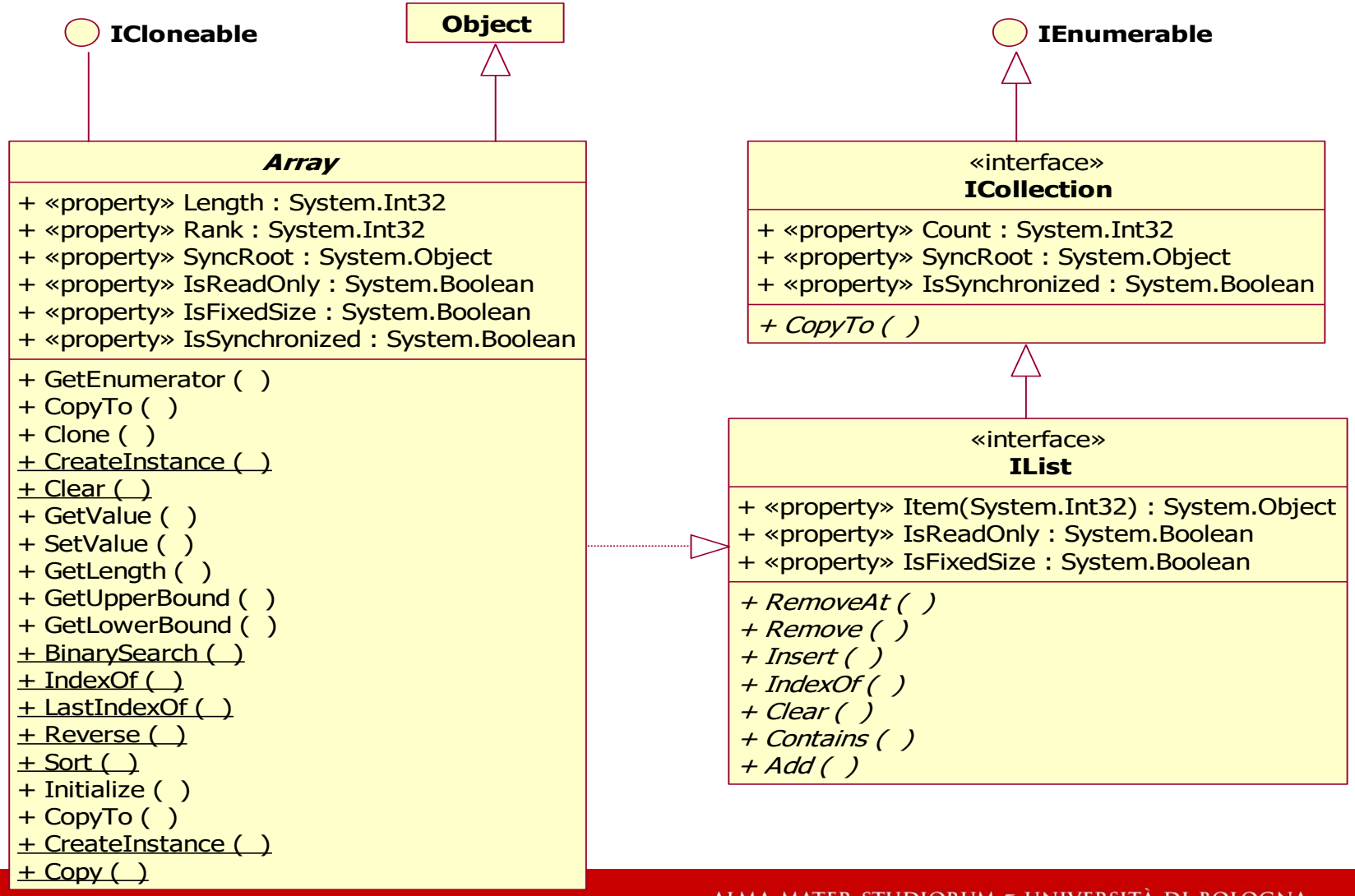
```
foreach (MyType obj in enumerable)
{
  …
}
```

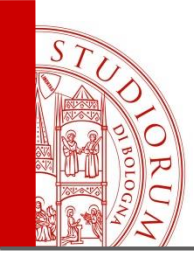
# System.Collections.IEnumerator

```
public class Contenitore : IEnumerable
{
  …
  public IEnumerator GetEnumerator()
  {
    return new Enumeratore(this);
  }

  class Enumeratore : IEnumerator
  {
    public Enumeratore(Contenitore contenitore) …

  }
}
```

**Esempio 1.6**

# System.Array

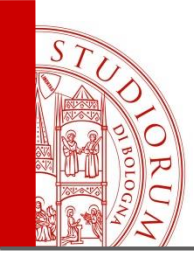




# System.Array

- Array mono-dimensionali

```
int[] a = new int[3];
int[] b = new int[] {3, 4, 5};
int[] c = {3, 4, 5};
// array of references
SomeClass[] d = new SomeClass[10];
// array of values (directly in the array)
SomeStruct[] e = new SomeStruct[10];
```



# System.Array

- Array multidimensionali (frastagliati)

```
// array of references to other arrays
int[][] a = new int[2][];
// cannot be initialized directly
a[0] = new int[] {1, 2, 3};
a[1] = new int[] {4, 5, 6};
```

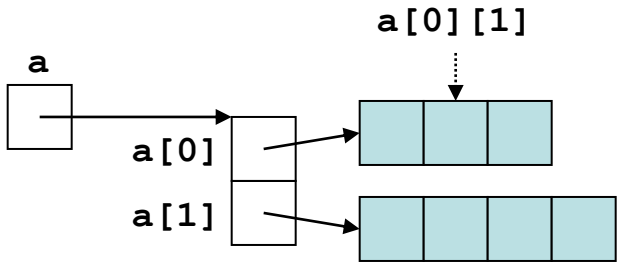
- Array multidimensionali (matrici)

```
// block matrix
int[,] a = new int[2, 3];
// can be initialized directly
int[,] b = {{1, 2, 3}, {4, 5, 6}};
int[,,] c = new int[2, 4, 2];
```

# System.Array

- **Frastagliati** (come in Java)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];
…
int x = a[0][1];
```




- **Matrici** (come in C, più compatti ed efficienti)

```
int[,] a = new int[2, 3];
…
int x = a[0, 1];
```